Towards a Configurable, Explainable, Scalable and Modular Rule-Based Expert System for Semantic Resource Matching in Graph Databases

1st Mirco Soderi
Data Science Institute
University of Galway
Galway, Ireland
mirco.soderi@universityofgalway.ie

2nd John Gerard Breslin
Data Science Institute
University of Galway
Galway, Ireland
john.breslin@universityofgalway.ie

2025 35th Irish Signals and Systems Conference (ISSC) | 979-8-3315-7593-9/25/$31.00 ©2025 IEEE | DOI: 10.1109/ISSC67739.2025.11291361

***Abstract*—This paper presents a modular open source rule-based expert system that leverages semantic web principles to evaluate relationships between entities stored in a graph database. The system is configurable by design, enabling users to define domain-specific rules that determine whether two resources match, based on user-submitted ontologies and associated graph data. The reasoning engine is entirely custom-developed and symbolic, based on definitions and conditions that are chained via logical expressions. Its modular architecture allows extensibility via Python modules for rule evaluation and data transformations, fostering adaptability to different domains. Semantic expressiveness is enabled through Neo4j and its neosemantics toolkit, facilitating ontology-driven CRUD operations and inference. The Web interface provides autogenerated forms derived from ontologies, simplifying data manipulation. Although general purpose, this system is particularly suited for sustainability-aware decision making in contexts where explainability, interoperability, and traceability are paramount, as it empowers users to configure logic aligned with dynamic sustainability criteria, supports transparent evaluation of alternatives based on heterogeneous semantic data, and delivers interpretable, traceable justifications for decisions involving ecological, societal, and economic dimensions. Compared to previous work, it uniquely integrates rule-based reasoning, modular extensibility, semantic graph storage, and explainable logic in a lightweight and deployable architecture.**

***Index Terms*—Expert systems, semantic web, ontology-driven systems, rule-based reasoning, graph databases, explainable AI, modular software architecture, sustainability decision support, Neo4j, FastAPI.**

I. Introduction

Expert systems are increasingly critical in domains where transparency, domain-specific configurability, and semantic interoperability are essential. Applications that support complex decisions in infrastructure planning, sustainability assessment, or social impact assessment often rely on heterogeneous data and require explainable outputs. To address these challenges, we present an open source, configurable, and modular expert system based on semantic graph technologies.

The proposed system enables users to define rules using domain-specific ontologies and to make decisions by evaluating whether a pair of resources in a graph database satisfies specified logical conditions. Rules are decomposed into *definitions* and *conditions*, combined via logical connectors and evaluated symbolically through Python functions. This design ensures full explainability, traceability, and extensibility. Ontology-based user interfaces are automatically generated, allowing intuitive creation and manipulation of both data and rules.

The system architecture is scalable and built using modern, lightweight technologies, such as FastAPI for the back-end, Neo4j with the neosemantics toolkit to store semantic graph data, and a simple JavaScript front-end for ontology-driven CRUD operations. Unlike prior systems, which often entangle rule logic with specific application code or rely on black-box AI models, our approach separates evaluation logic into modular, reusable Python components. This promotes adaptability, modularity, and reusability in all sectors.

From a sustainability perspective, such a system allows practitioners to model context-sensitive rules, environmental, social, or technical, and to evaluate potential actions in a transparent and justifiable way [1]–[3]. Although this work is applicable across domains, its value is particularly clear in decision support scenarios where semantic clarity and auditability are critical.

The remainder of the paper is structured as follows. Section II discusses related work. Section III details the architecture of the system. Section IV presents the reasoning engine. Section V describes the ontology-driven user interface. Finally, Section VI concludes and outlines future directions.

II. Related Work

The proposed system is at the intersection of expert systems, semantic web technologies, and graph-based knowledge repre-

This publication has emanated from research supported in part by the European Digital Innovation Hub Data2Sustain, co-funded by Ireland's National Recovery and Resilience Plan (the EU's Recovery and Resilience Facility), the Digital Europe Programme, and the Government of Ireland, and by a grant from Taighde Éireann - Research Ireland under Grant Number 12/RC/2289_P2 (Insight). Chat-GPT has been used during the production of the initial version of this manuscript, whereas Writefull has been used to refine the final version.

sentation. Previous work in rule-based reasoning has produced well-known tools such as CLIPS [4] and Drools [5], but these are often tightly coupled with specific data representations and lack semantic awareness. More recent frameworks such as Apache Jena [6] and RDF4J [7] support semantic querying via SPARQL, but are generally focused on RDF stores and do not integrate well with property graph models like Neo4j.

Several projects have explored semantic data integration for environmental or urban planning contexts, which shows the relevance of ontology-based systems for sustainability. However, these systems often rely on static rule definitions or focus exclusively on reasoning over prestructured datasets. In contrast, the proposed system empowers domain experts to define and extend rules at run-time, using ontologies as both data schema and user interface generation tools.

Neo4j has been adopted in semantic contexts through its neosemantics toolkit, which enables the import and query of RDF statements [8]. Although Neo4j AI capabilities, including link prediction and graph embeddings, offer promise for inductive reasoning, the proposed system adopts a purely symbolic approach. This ensures explainability and control, key requirements in domains such as public infrastructure or regulatory compliance.

In terms of architecture, modularity has received increasing attention in knowledge-based systems [9]. The proposed system advances this trend by isolating transformation logic and condition evaluation into Python modules, facilitating easy customisation and reuse. The reasoning engine itself is designed to be domain-agnostic and interpretable, which resonates with recent calls for transparent AI in high-stakes decision-making.

To our knowledge, no previous open-source system provides a lightweight, scalable, and fully modular expert system that combines rule-based logic, semantic graph data, and ontology-driven UI in a web-accessible package. This paper aims to fill this gap.

III. System Architecture

The proposed system is a modular open source expert system designed to support configurable and explainable reasoning over semantic graph data. It is implemented as a lightweight containerised web application composed of a FastAPI-based back-end, a static JavaScript front-end, and a Neo4j graph database extended with semantic capabilities via the neosemantics toolkit. Figure 1 presents a high-level overview of the system architecture.

A. System Components

The system consists of three main components:

- **Frontend:** A static web interface served by FastAPI using `StaticFiles`. It is implemented with HTML, CSS, and JavaScript (using jQuery), and provides user interaction for model upload, data manipulation, rule configuration, and decision requests.
- **Back-end:** A set of RESTful APIs developed with FastAPI. The APIs support ontology submission, CRUD operations for data and rules, and most importantly, the reasoning engine, which includes the evaluation of the configured definitions and the verification of the configured conditions. All interactions are handled from the back-end, using the Neo4j Python driver.
- **Graph Database:** A Neo4j instance extended with the neosemantics (n10s) toolkit to support RDF import and semantic querying. The database also uses the APOC plugin for initialisation and indexing. Ontologies and all graph data, including the system configuration (definitions and rules), are stored in this database.

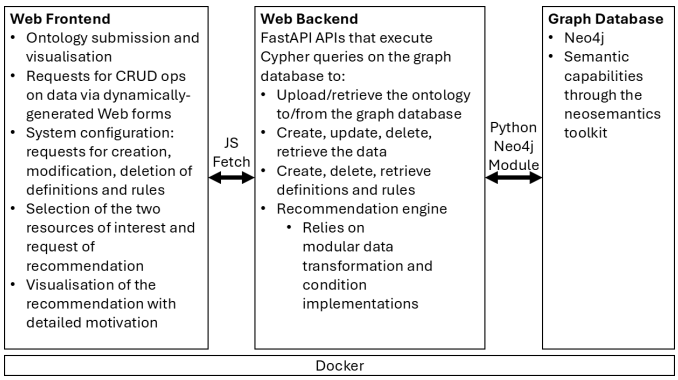

Fig. 1. System Architecture Overview

B. Ontology Management

Ontology files can be uploaded through the front-end and are stored in a server-accessible static directory. The back-end then issues a Cypher query to import the ontology using the `n10s.onto.import.fetch` procedure, passing the ontology URL and the serialisation format specified by the user. Once loaded, the ontology informs both the structure of the data and the generation of dynamic input forms on the front-end.

Although the ontology is assumed to be stable post-upload, the front-end reflects any direct modifications made in Neo4j (e.g., via Cypher queries), making the system highly responsive to ontology changes.

C. Reasoning Engine

When a recommendation about whether two data resources match is requested, the reasoning engine performs a server-side evaluation based on a sequence of well-defined steps.

1) The definitions are retrieved from Neo4j. Each definition is of type *literal*, *property*, or *computation*, with corresponding parameters.
2) The value of each definition is computed:
   - *Literal:* A user-specified constant.
   - *Property:* Retrieved via Cypher traversal starting from either selected node.
   - *Operation:* Result of applying a Python function (from `operations.py`) to the value of another definition.

3) Rules are retrieved as fragment arrays; a fragment can contain a condition, bracket, logical operator, or a reference to a definition. Fragments are rewritten and then joined into a Boolean Python expression.
4) The expression is evaluated based on Python condition implementations from the `conditions.py` module.
5) If all rules return `True`, the system returns a *match*; otherwise, a *mismatch*.

The system returns not only the final recommendation, but also a full JSON structure representing the evaluated definitions, rules, and condition outcomes, ensuring complete transparency of the decision logic.

D. Data and Rule Storage

Both rules and definitions are stored directly in Neo4j:

- The definitions are labelled as `Rule` and `ExpertSystemRuleDefinition`, with parameters stored as properties (`definiendum`, `definiens_type`, and up to five `definiens_value_X` fields).
- Rules are labelled as `Rule` and `ExpertSystemRuleCondition`, with a `name` and a `condition` description represented as an ordered set of fragments.

Front- and back-end communicate through RESTful calls.

E. Modularity

The system is modular in design. Data transformations and conditions are implemented in two separate files: `operations.py` and `conditions.py`, respectively. These are invoked dynamically via name-based dispatch and evaluated within a controlled context. While non-developers cannot yet upload or register new functions, the modular design supports easy extension by developers.

F. Deployment and Interoperability

The full system is containerised using Docker Compose, comprising two services: `application` (FastAPI back-end and Web front-end) and `database` (Neo4j with APOC and neosemantics). All artefacts are publicly available on GitHub[1].

Thanks to its RESTful architecture, the system is interoperable with other platforms and services, supporting integration into broader decision workflows or semantic data infrastructures.

IV. Reasoning

The core of the system is a custom-built reasoning engine that determines whether two user-selected entities in the semantic graph satisfy a configurable set of conditions defined via a modular symbolic rule system. The engine prioritises full explainability, deterministic logic, and extensibility through user-defined definitions and rules.

[1]https://github.com/mircosoderi/rule-based-semantic-match

A. Reasoning Workflow

When a match request is sent, the front-end calls the `/api/recommendation` endpoint with two parameters: `left` and `right`, corresponding to the identifiers of the two selected nodes. All reasoning is performed server-side, and the front-end is responsible only for initiating the request and rendering the results.

The reasoning pipeline comprises the following phases:

1) **Definition Retrieval:** The system queries the Neo4j database to retrieve all defined definitions by the user. Each definition is labelled as a node with metadata indicating its name (`definiendum`), type (`definiens_type`), and up to five positional parameters (`definiens_value_0` through `definiens_value_4`).
2) **Definition Evaluation:** The engine computes the value of each definition depending on its type:
   - *Literal:* A static value, specified directly in the definition parameters.
   - *Property:* Retrieved via a Cypher query that navigates the graph from one of the two selected nodes; parameters specify the start node (left or right), its expected semantic class and the path to the value.
   - *Operation:* The name of a function and its operand (another definition) are specified. The function is dynamically loaded from the `operations.py` module and applied to the operand value.
3) **Rule Retrieval:** Rules are represented as nodes in the Neo4j database, in which the `condition` property carries an ordered set of fragments, each of which contains a reference to a definition, names of a condition, a logical operator, or a bracket.
4) **Rule Rewriting:** The system parses each rule's fragment array to construct an evaluatable Python expression. When a condition fragment is encountered (e.g. "is equal to"), it is replaced by a function call (e.g. "is_equal_to(val_l, val_r)") referencing the value(s) adjacent definition(s) as parameters. All condition functions reside in the `conditions.py` module. Definition references are replaced with empty strings (they are incorporated in function calls anyway), and the logical operators and brackets are preserved.
5) **Rule Evaluation:** The resulting expression string (e.g., `is_equal_to("value1", "value2") and is_not_empty("value3")`) is evaluated using Python's `eval()` function in a safe context. The output is a Boolean result that indicates whether the rule is satisfied.
6) **Final Decision:** If all rules are evaluated to `True`, the system returns a `match`; otherwise, a `mismatch`.

B. Reasoning Workflow Example

In this example, we consider the case in which:

- The user has selected as left resource a resource that represents a person named Mario.

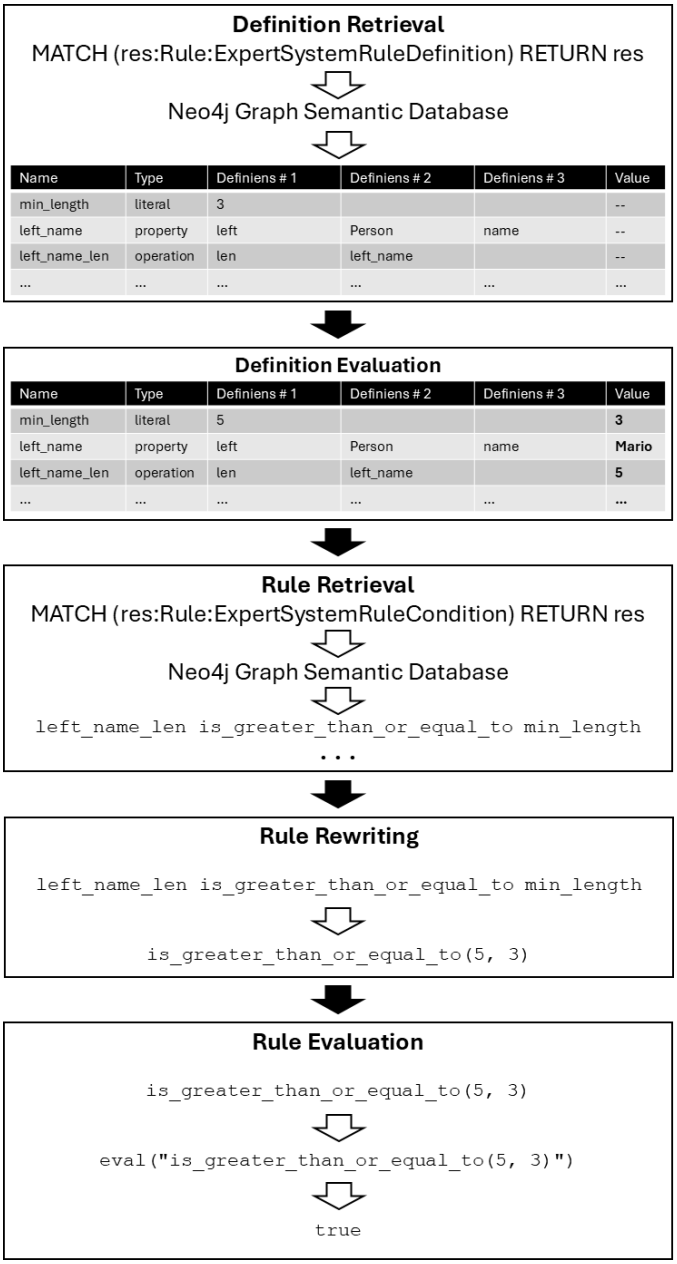

Fig. 2. Example of a simple reasoning workflow

- There is a rule that dictates that the name must be at least 3 characters long.
- There is an operation function named *len* that returns the length of the string received in the input.
- There is a condition function named *is_greater_than_or_equal_to* that returns true if the first argument is greater than or equal to the second.

In this case, the reasoning would develop as shown in Fig. 2.

C. Explainability and Output Structure

In addition to the final recommendation, the system returns a comprehensive motivation containing all definitions with their types, parameters, and computed values, along with all rules with their definition and the result of the evaluation of each condition in each rule.

This structured response allows users and systems to fully trace and interpret the decision-making process, satisfying key explainability requirements for transparency and auditability.

D. Performance Considerations

Although the system does not implement explicit caching, Neo4j internal query caching enhances performance for repeated lookups. To optimise query speed, each node in the graph is assigned an `eid` property (equal to Neo4j internal `elementId`) during creation, and indexes are built on this property. All property-based definitions use labels and relationship types in Cypher queries to further improve performance in high-connectivity graphs.

E. Extensibility

The symbolic and modular nature of the reasoning logic enables developers to extend the system by adding new operations to `operations.py` and new condition functions to `conditions.py`.

V. User Interface

The user interface (UI) of the system is designed to be lightweight, semantically aware, and functionally aligned with the principles of explainable and configurable expert systems. Although intentionally minimalist, the front-end provides full access to the ontology-driven structure, supports dynamic interaction with semantic data, and visually conveys the logic behind decision-making processes. Figure 3 and Figure 4 give a sense of how the recommendation page looks like.

A. Interface Overview

The UI is divided into four main sections, accessible through a top-level navigation menu.

- **Model**: Uploads and visualises the data model (ontology).
- **Data**: Displays and edits semantic data instances using autogenerated forms.
- **Rules**: Allows users to define and manage decision rules via guided input.
- **Decision**: Enable pairwise node comparison and visualise rule-based reasoning outcomes.

Each section is rendered client-side using HTML, CSS, and jQuery. Static assets are served by the FastAPI back-end through a `/html` mount point.

B. Ontology-Driven Form Generation

When the data page loads, the front-end sends a `GET` request to the `/api/model` endpoint. The server responds with a JSON representation of the ontology, which consists of `Resource` nodes and relationships retrieved from the database, where they were stored by the neosemantics import. It is worth mentioning that semantic classes, properties, and relations are all stored in the database as nodes with appropriate labelling.

The front-end identifies semantic classes as nodes labelled `n4sch__Class`, and for each class:

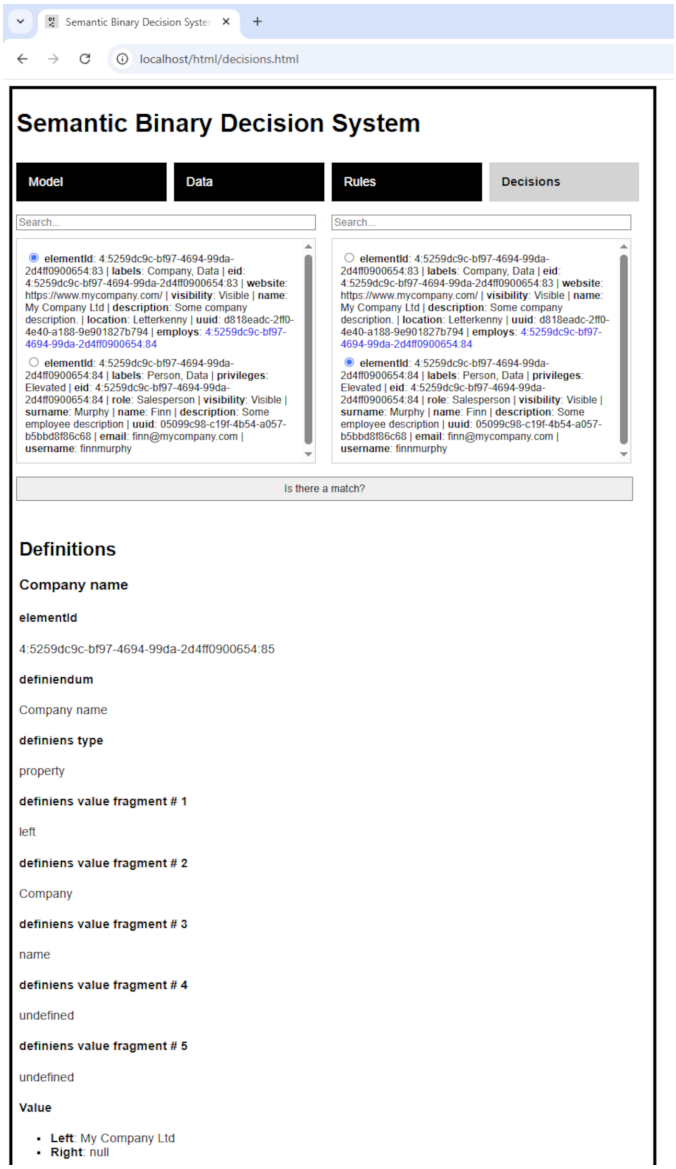

Fig. 3. User interface overview: screenshots of the model, data, rules, and decision pages.

1) The heading is appended with the class name.
2) An update/delete form is dynamically generated, where all existing instances of that specific class are listed.
3) A second form is dynamically generated to input new instances, with fields generated using the defined properties and relationships of the ontology (Figure 5). In particular, the UI navigates `n4sch__DOMAIN` and `n4sch__RANGE` relationships to extract the relevant fields for each class, and input field types are chosen according to the associated semantic type (e.g., text, number, select). Relationship fields are presented as lists of checkboxes populated from data nodes of the expected range class.

Fig. 4. User interface overview: screenshots of the model, data, rules, and decision pages.

C. Rule Editor Interface

The rule configuration interface is fully drop-down driven. Users build a rule by iteratively selecting elements from a uniform drop-down menu that includes definitions, conditions, logical connectors, and brackets.

Each time a selection is made, a new drop-down appears to guide the user through the rule composition. While currently agnostic to rule syntax, the system assumes validity and saves the rule as-is. Rules are only validated indirectly during evaluation; any invalid rule is marked as "not satisfied", contributing to a "mismatch" decision.

D. Recommendation Page and Visual Feedback

When the user selects two data nodes and triggers a recommendation request, the user interface displays a detailed evaluation summary that includes the full list of definitions with their computed values, and a structured list of rules with pass/fail results for each rule.

Unsatisfied rules are rendered in red, bolded, and enlarged text to aid in quick identification. Although not formatted with styled visual components, the output is clean, clearly segmented, and easy to interpret. As such, it provides complete transparency in the reasoning process.

E. Limitations and Future Work

The interface is minimalist but semantically rich, with a fixed-width layout. Basic input validation is performed through appropriate HTML markup and JavaScript based on property types (e.g. integers, booleans). However, the UI currently lacks

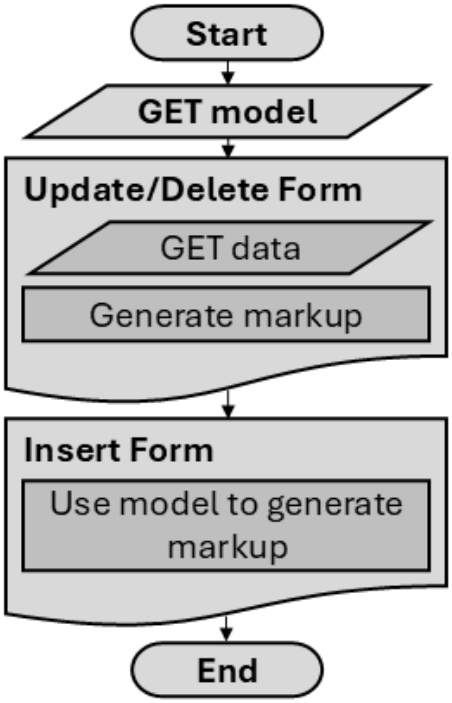

Fig. 5. Ontology-driven form generation: from semantic classes, properties, and relations to HTML markup for input fields.

real-time synchronisation, accessibility features, and responsive behaviour.

The planned improvements include the following.

- Live syncing of ontology and data updates using WebSockets.
- Smarter, context-aware rule editors to prevent invalid syntax at input time.
- Support for front-end ontology editing and re-importing.
- A redesigned responsive interface with accessibility support.
- User roles and authorisation mechanisms.
- Possibility to register, package, or load modules (conditions, operations) dynamically, possibly even from external sources, and to auto-generate user interface through introspection and related techniques.

The current interface strikes a balance between functionality and simplicity, allowing users to interact with semantically modelled data and configure decision logic with minimal effort and maximal clarity.

VI. Conclusion and Future Work

This paper presents a modular, configurable, and explainable rule-based expert system grounded in semantic web principles and graph databases. Users can model domain-specific reasoning using ontologies and declarative rules to determine matches between graph nodes. The system pairs a lightweight FastAPI backend with a semantically aware front end that supports ontology-driven data manipulation, rule configuration, and decision explanation. It uses a fully symbolic, transparent reasoning engine based on rule rewriting, ensuring traceable and auditable outcomes.

Prioritising explainability and modularity, the system offers a clean and extensible interface to enhance decision logic. Unlike black-box AI, it supports transparency and semantic interpretability, making it suitable for domains such as sustainability assessment and policy planning. The application is open-source and containerised, easing integration into broader systems.

Future work includes support for dynamic condition registration, remote libraries, context-aware validation, ontology editing, and live rule tracking via real-time sync (e.g. WebSockets). Usability improvements and the creation of domain-specific logic packs will also promote reuse and accessibility.

This system provides a foundation for semantic decision tools that are both explainable and extensible, enabling interoperable user-centric reasoning platforms.

Appendix A
How AI assisted in the preparation of this work

ChatGPT (GPT-4-turbo) [10], a large language model developed by OpenAI, was extensively used for the preparation of the initial draught of all sections of this manuscript. The phrasing was then finalised with the help of Writefull [11], the integrated AI assistant in Overleaf [12]. The ChatGPT chat showing how the tool was used for the preparation of this work is available at https://chatgpt.com/share/67f7f3ac-b2bc-8013-852d-ad0ce2b9cfcc

References

[1] C. Iddianozie and P. Palmes, "Towards smart sustainable cities: Addressing semantic heterogeneity in building management systems using discriminative models," *Sustainable Cities and Society*, vol. 62, p. 102367, 2020.

[2] S. Bischof, A. Karapantelakis, C.-S. Nechifor, A. P. Sheth, A. Mileo, and P. Barnaghi, "Semantic modelling of smart city data," 2014.

[3] L. J. Ramirez Lopez and A. I. Grijalba Castro, "Sustainability and resilience in smart city planning: A review," *Sustainability*, vol. 13, no. 1, p. 181, 2020.

[4] G. Riley, "Clips: A tool for building expert systems," NASA Johnson Space Center, Tech. Rep. JSC-23762, 1989, available at https://www.clipsrules.net.

[5] Red Hat, Inc., *Drools: A Rule Engine for Complex Event Processing and Business Logic Integration*, 2024, available at https://www.drools.org.

[6] J. J. Carroll, I. Dickinson, C. Dollin, D. Reynolds, A. Seaborne, and K. Wilkinson, "Jena: Implementing the semantic web recommendations," in *Proceedings of the 13th International World Wide Web Conference (WWW2004) - Alternate Track Papers & Posters*. ACM, 2004, pp. 74–83.

[7] J. Broekstra, A. Kampman, and F. van Harmelen, "Sesame: A generic architecture for storing and querying rdf and rdf schema," *The Semantic Web–ISWC 2002*, vol. 2342, pp. 54–68, 2002.

[8] Neo4j Labs, "Neosemantics (n10s): Neo4j rdf and semantics toolkit," https://neo4j.com/labs/neosemantics/, 2025, accessed: 2025-05-15.

[9] M. d'Aquin, A. Schlicht, H. Stuckenschmidt, and M. Sabou, "Criteria and evaluation for ontology modularization techniques," *Modular Ontologies: Concepts, Theories and Techniques for Knowledge Modularization*, pp. 67–89, 2009.

[10] OpenAI, "Chatgpt (gpt-4-turbo)," https://chat.openai.com, 2024, accessed: 2025-04-10.

[11] Writefull, "Writefull – ai-based language feedback tool," https://writefull.com, 2025, accessed: 2025-04-10.

[12] Overleaf, "Overleaf – online latex editor," https://www.overleaf.com, accessed: 2025-04-10.